

NOTE: In any of these questions where types' sizes is relevant, you should assume the following sizes:

char	1 byte	pointer	4 bytes	long int	8 bytes
short int	2 bytes	int	4 bytes	double	10 bytes

1. [15] Suppose you want to implement sets of uppercase letters 'A'.. 'Z' in C++ in a way that minimizes the storage requirements.

a. Give definition of a type LetterSet.

```
typedef int LetterSet;
```

b. Define the set-membership function `bool mem(char, LetterSet)`.

```
bool mem( char c, LetterSet S )
{
    if( 'A' <= c && c <= 'Z' )
        return (S >> (c - 'A')) & 1;
    return false;
}
```

c. Define the element-addition function `LetterSet add(char, LetterSet)`.

```
LetterSet add( char c, LetterSet S )
{
    if( 'A' <= c && c <= 'Z' )
        return (S | (1 << (c - 'A')));
    return S;
}
```

2. [10] Translate the following C/C++ function, which uses array indexing, into an equivalent one which uses explicit pointers and no array indexing. The function's type should *not* be changed, and no integer counter should be used.

```
int runLength( const double a[], int i, int k )
{
    double t = a[i];
    i++;
    while ( i < k && a [i] >= t) i++;
    return i;
}
```

A solution:

```
int runLength( const double a[], int i, int k )
{
    double* dp = a+i;
    double* dpLast = a+k;
    double t = *dp;
    dp++;
    while (dp < dpLast && *dp >= t) dp++;
    return dp-a;
}
```

3. [10] Define a Haskell algebraic type `Position` which represents a point in a plane using two `Float` values. Provide for two coordinate systems:

- Cartesian (x,y) coordinates
- polar (r,θ) coordinates

Use field names to remind users of the fields' significance.

Include in your answer code so that if `p1, p2 :: Position` then `p1 == p2` is `True` iff `p1` and `p2` represent the same position.

```
> data Position = Cart { x,y :: Float } | Polar { r,theta :: Float }
> cart :: Position -> [Float]
> cart (Cart x y) = [x,y]
> cart (Polar r theta) = [r * cos theta, r * sin theta]
> instance Eq Position where
>   p1 == p2 = cart p1 == cart p2
```

4. [10] Given the following Haskell script:

```
g x = 3 * f x
h = k 10
  where
    f z = 8 * z
    k y = g (y*5)
f x = x+2
```

- a. Circle the value of `h`

156 1200

- b. What change in Haskell would cause `h` to have the value you did not circle?

The binding of free variables would have to change from static to dynamic.

5. [15] Predict the output of the following code

```
integer array A = [1,2,4,8,16,32];
integer x = 1;
integer linus( integer a, b ) {
  a := a+2;
  b := b*2;
  return 100*A[x] + 10*b + a;
}
main() {
  print( linus( x, A[x] ) );
}
```

assuming the smallest array index is 0, and assuming that arguments are bound to parameters

- a. by reference

843 $(100*8 + 10*4 + 3)$

- b. by value

243 $(100*2 + 10*4 + 3)$

- c. by name

1763 $(100*16 + 10*16 + 3)$

6. [10] Give the values of *a* and *b* after the execution of the following C code:

```
int a = 0; int b = a + (a && (a = 2)) + a++;
```

Show how these values are determined, and assume that (i) any side effects take place immediately, and (ii) whenever the order of operand evaluation is not specified in the language definition, it is

- a.** left to right $b = a + (a \ \&\& \ (a = 2)) + a++$ **where** $a = 0$
 $\rightsquigarrow b = 0 + (a \ \&\& \ (a = 2)) + a++$ **where** $a = 0$
 $\rightsquigarrow b = 0 + (0 \ \&\& \ (a = 2)) + a++$ **where** $a = 0$
 $\rightsquigarrow b = 0 + 0 + a++$ **where** $a = 0$
 $\rightsquigarrow b = 0 + a++$ **where** $a = 0$
 $\rightsquigarrow b = 0 + 0$ **where** $a = 1$
 $\rightsquigarrow b = 0$ **where** $a = 1$
- b.** right to left $b = a + (a \ \&\& \ (a = 2)) + a++$ **where** $a = 0$
 $\rightsquigarrow b = a + (a \ \&\& \ (a = 2)) + 0$ **where** $a = 1$
 $\rightsquigarrow b = a + (1 \ \&\& \ (a = 2)) + 0$ **where** $a = 1$
 $\rightsquigarrow b = a + (1 \ \&\& \ 2) + 0$ **where** $a = 2$
 $\rightsquigarrow b = a + 1 + 0$ **where** $a = 2$
 $\rightsquigarrow b = 2 + 1 + 0$ **where** $a = 2$
 $\rightsquigarrow b = 3 + 0$ **where** $a = 2$
 $\rightsquigarrow b = 3$ **where** $a = 2$

7. [15] You are given this pair of Ada tasks:

```
task zeros is
begin
  loop
    synch.zero();
    put( 0 );
  end loop;
end zeros;
```

```
task ones is
begin
  loop
    synch.one();
    put( 1 );
  end loop;
end ones;
```

where `put(n)` appends the value of *n* to the standard output stream.

Define the body of a task `synch` so that the difference between the number of 1's and the number of 0's that these two tasks have appended to the standard output stream never exceeds *d*, where the value of *d* is provided to `synch` in an initialization call.

Also provide for a call to `synch.howMany` by which a caller can learn the total number of 1s and 0s that these two tasks have appended to the standard output stream.

```
task body synch is
  n1s, n0s, ns, diff : integer;
begin
  n1s := 0;
  n0s := 0;
  ns := 0;
  accept init( d : in integer ) do
    diff := d;
  end init;
  loop
    select
      when n0s-n1s < d =>
        accept zero do
          n0s := n0s + 1;
          ns := ns + 1;
        end zero;
      or
      when n1s-n0s < d =>
        accept one do
          n1s := n1s + 1;
          ns := ns + 1;
        end one;
      or
        accept howMany( n : out integer )
          n := ns;
        end howMany;
    end select;
  end loop;
end synch;
```